



PROGRAMMING Expert

Your music collection is in disarray and you need a comprehensive database of your tunes. Mike James explains how some judicious programming and a free open-source CD database resource can help

PROGRAMMING NEWS

TOP PROGRAMMERS WORK FOR POPULAR SEARCH ENGINES

If you've ever wondered why search engines keep getting better while other software doesn't, it's probably because all the best programmers have decided to work for Yahoo! and Google. A recent report in *Business Week* commented on the high number of talented software developers leaving Microsoft, Amazon, Sun and eBay to join the quest for a better search engine.

The attraction is said to be the excitement and action that these companies provide. If you work for Google, you can spend a day a week working on your pet project, and relax playing roller hockey in the underground garage or racing remote-controlled blimps through the huge office space. Other perks include the gourmet meals served in the company cafeteria.

NEW OPEN-SOURCE OPERATING SYSTEM GETS BOOST

FreeDOS, an open-source equivalent of the MS-DOS operating system, is close to its first non-beta release. It aims to be completely compatible with the real thing while still running on a wide range of hardware including old and low-powered PCs. As well as being an operating system, FreeDOS includes many familiar utilities and development languages.

Until recently, FreeDOS lacked a graphical user interface (GUI) and was mostly of interest to programmers developing embedded systems such as robotics or control. But it will benefit from the arrival of OpenGEM Release 4, which is based on the DR GEM GUI popular in the 1980s before Windows became the dominant force. OpenGEM is a single-tasking 16-bit GUI. It won't challenge Windows or the Linux desktops, but it certainly makes FreeDOS more useful. See www.freedos.org and <http://gem.shaneland.co.uk> for details.



We recently decided to create a database to keep track of all our music CDs. We wanted the database to tap into one of the websites that look up the details of a disc that is placed in a CD drive. This makes it possible to build a complete list of discs, including all their details, without much effort.

The best-known online CD database is Cddb. Until recently this was open source and free, but it has changed its name to Gracenote and started charging programmers who sell programs that use it. If you are creating software on a non-profit basis then you can apply for a free licence, but there are forms to fill in and approvals to obtain. When your application is working, you also have to submit it to Gracenote for validation.

Fortunately, there is still a free open-source CD database called Freedb. This is a lot less trouble to use and seems to find all the obscure CDs with which I've tested it. Visit www.freedb.org for details.

Unfortunately, Freedb doesn't have an advanced interface. There's no associated web service and it won't return its data using XML, so quite a lot of the work involved in using it entails writing custom code to communicate with it.

We also need to access information stored in the table of contents (TOC) of an audio CD. This is used to compute a kind of fingerprint for the disc, which is then looked up in the database. The simplest way of doing this is to work with the Media Control Interface (MCI) API. This is one of the oldest of the Windows multimedia APIs, but it's still essential as it hasn't been replaced by a COM object or a .NET class. While working with the MCI API, it is also worth discovering how to play a CD, simply because it's easy and useful.

We wrote this program in C# using the C# Express Beta 2, which can be downloaded free from the Microsoft website. But you could easily convert the examples to almost any language that can call the Windows API and make a TCP/IP connection.

Topics covered this month also include making API calls in C#, creating class wrappers for API

functions, making a TCP/IP connection in C# and creating a TCP/IP client.

CHANGE THE RECORD

If you want to control a multimedia device, you might think of using DirectX or going in search of a .NET class that does the job for you. But there are no facilities in DirectX or .NET that allow you to control a CD drive. To do this you have to use the old Windows API, the MCI. If you use Visual Basic, you might have accessed the MCI via the MCI ActiveX control, but you can do the same job by directly calling the API.

The .NET environment doesn't like you to use direct API calls because they make the code unmanaged, but the lack of facilities for multimedia in the Framework classes makes it unavoidable here. To keep the unmanaged code to a minimum, we'll wrap the MCI calls that we use in a class.

Start a new Windows project and use the Project, Add Class command to add a class named CCDAudio. As we are going to call an API function, we need the help of the Framework Interop classes. Add the using statement:

```
using System.Runtime.InteropServices;
```

Next we need a definition of the API function that we want to call. In this case, the only MCI function we need is `mciSendString`:

```
class CCDAudio
{
    [DllImport("winmm.dll")]
    extern static int mciSendString(
        string command,
        StringBuilder Buffer,
        int bufferLength,
        int nothing);
}
```

The `DllImport` statement specifies the name of the DLL file that the API function is in and the name of the function and the parameters it expects. There is



often some flexibility in doing this because the Framework Interop classes perform conversion between appropriate types. In this case, a `StringBuilder` object is being used rather than a byte array buffer, because it's a more convenient way to receive the result of the API call. You can set up any API calls that you need in a similar way.

To use the `mciSendString` function, you send a command string to specify what you want to do and get the results back in the `StringBuilder` buffer. You can see the range of MCI commands at http://msdn.microsoft.com/library/en-us/multimed/html/_win32_mci.asp.

We need two global variables to use `mciSendString`: the response buffer and an error code:

```
private int err;
private StringBuilder response =
    new StringBuilder(128);
```

The class can use its constructor to open the `cdaudio` device and its destructor to close it:

```
public CCDAudio()
{
    err = mciSendString("open cdaudio",
        response, 128, 0);
}
~CCDAudio()
{
    err = mciSendString("close all",
        response, 128, 0);
}
```

No attempt at error handling has been made. It is up to you to add instructions that deal with a non-zero error code being returned in error.

SOUND BASIS

Now we have the basics for opening the `cdaudio` device, we can use it. By studying the available MCI commands, you can create methods for the `CCDAudio` class that play, stop, pause, eject and seek a given track:

```
public int play()
{
    return err = mciSendString("play
        cdaudio",
        response, 128, 0);
}
public int stop()
{
    return err = mciSendString("stop
        cdaudio",
        response, 128, 0);
}
public int pause()
{
    return err = mciSendString("pause
        cdaudio",
        response, 128, 0);
}
public int eject()
{
    return err = mciSendString("set
        cdaudio door open",
        response, 128, 0);
}
public int seek(int track)
```

```
{
    err = mciSendString("set cdaudio
        time format tmsf",
        response, 128, 0);
    if (err != 0) { return err; }
    err = mciSendString("seek cdaudio
        to " +
        track.ToString(), response, 128,
        0);
    return err;
}
```

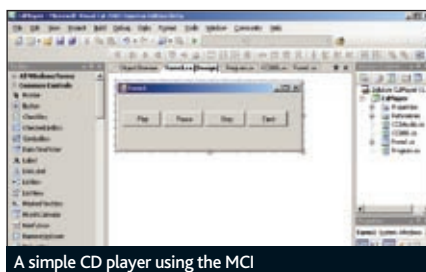
In each case, these methods handle errors by simply returning the error code for the calling program to resolve. The only complicated method is `seek`; this allows you to specify a track number to which you jump. The `seek` method sets the CD player into a 'track, minutes, seconds, frame' format and sends the track number you requested. You could send a more accurate location by including minutes, seconds and frames. A frame is the basic unit in which an audio CD works, and it represents $\frac{1}{75}$ th of a second of sound. If you understand these methods you can add other functions by reading the MCI documentation, and even create new classes to wrap other devices such as video players.

To show how you would use the class, we'll place four buttons on a form with the captions Play, Pause, Stop and Eject. We create an instance of the `CCDAudio` class:

```
CCDAudio cdaudio;
public Form1()
{
    InitializeComponent();
    cdaudio = new CCDAudio();
}
```

The button click event handlers are very simple, calling the methods provided by the object:

```
private void button1_Click(object
    sender, EventArgs e)
{
    int err=cdaudio.play();
}
private void button2_Click(object
    sender, EventArgs e)
{
    int err = cdaudio.pause();
}
private void button3_Click(object
    sender, EventArgs e)
{
    int err = cdaudio.stop();
}
private void button4_Click(object
    sender, EventArgs e)
{
    int err = cdaudio.eject();
}
```



A simple CD player using the MCI

```
int err = cdaudio.eject();
}
```

No attempt has been made to handle errors, but you should find you can start, stop and eject a CD. This could form the start of a CD player program.

ON THE RIGHT TRACK

The MCI makes it easy to create an audio player, but we want it to get the necessary data to look up a disc in an online CD database. The information needed is the number of tracks; the exact starting time of each track, measured in frames; and the total playing time of the disc in seconds. Getting the number of tracks and the start time of each track is easy as MCI supports these operations. Finding the total playing time of the disc is harder as it has to be worked out from other data.

The simplest way is to add a new method called `getDiscData` to the existing `CCDAudio` class. This will be designed to return a string formatted as:

```
<tracks> <start of track 1> <start
of track 2> ... <start of final track>
<total playing time>
```

This isn't a general format, but it makes looking up the disc easy. The first part of the method simply gets the number of tracks on the disc after setting the MCI into minute, second, frame format:

```
public string getDiscData()
{
    StringBuilder command = new
        StringBuilder(256);
    err = mciSendString("set cdaudio time
        format msf",
        response, 128, 0);
    if (err != 0) { return err.ToString();
    };
    err = mciSendString("status cdaudio
        number of tracks",
        response, 128, 0);
    if (err != 0) { return err.ToString();
    };
    int tracks = Convert.ToInt32(response.
        ToString());
    command.Append(tracks+" ");
}
```

The return value is built up using a `StringBuilder`. Again, the error handling is left for the calling program to deal with. The new method simply returns the error code as a string.

Now we know the number of tracks, we simply use the MCI API to query the starting position of each one in turn. The starting position of each track is also saved in an array called `offsets`:

```
int[] offsets = new int[tracks];
string[] temp;
int minutes, seconds, frames;
```

Getting the starting position of each track is easy:

```
for (int i = 1; i <= tracks; i++)
{
    err = mciSendString("status cdaudio
        position track " +
        i.ToString(), response, 128, 0);
```

The response string now contains the starting position formatted as minutes:seconds:frames. The



REVIEW BOOK

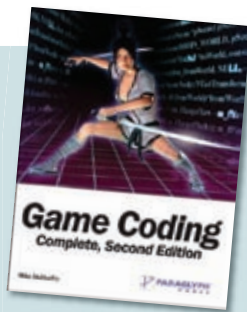
Game Coding: Complete, Second Edition

RATING ★★★★★

PRICE £31

ISBN 1932111913

PAGES 904



You may think this is yet another book on game programming, but Mike McShaffrey's approach is different. It's in C++ and mostly uses DirectX, but it's far more than a step-by-step introduction to creating simple games. It is clear the writer has experienced the real problems of creating big games. He explains how you go about programming and refers to silly mistakes he has made. He describes how he took to object-oriented programming so completely that he managed to choke the compiler with all the derived classes he created. The project became so unstable, his team was frightened to add new variables and resorted to coding values into the unused high bytes of existing variables.

There are code examples and discussions of the basics, but this isn't a beginner's book. You need to know C++, have some idea of how to create a 2D/3D game and understand the basics of sprites, meshes, textures and so on. You don't have to be interested in creating a game, though. The stories are sufficiently scary to appeal to anyone who is working on a large project.

This is a huge work and you're unlikely to read it all. But if you want to hear from someone who's been there and done that, it's an enjoyable read.

✓ The author has years of experience working on big projects

✗ If you don't know C++ or games programming, this won't help

REVIEW BOOK

Sams Teach Yourself PHP in 10 Minutes

RATING ★★★★★

PRICE £11

ISBN 0672327627

PAGES 256



PHP is a scripting language for web pages. It is easy to use, powerful and free, and this accounts for the increasing number of web pages that you encounter that end in .php. It's a viable alternative to both Java and ASP.NET. It's also a server side script so it can integrate with databases on the server.

Linda Barr and Chris Newman's book is a short introduction to PHP – but 10 minutes? Why are Sams' *Teach Yourself* books, which can be great, always ridiculously optimistic about the time you need to learn any topic?

This book covers the basics and shows you how to master form validation, cookies, user authentication and so on. There is even a chapter on the MySQL open-source database. The book has two parts: language facilities, arrays, classes and control constructs, and applications of the language. We thought essential details of installing PHP were omitted, but it's covered in an appendix. This is the weakest point as far as Windows users are concerned; open-source software can be confusing due to the lack of slick installation procedures.

Still, if you're looking for a pocket guide to PHP, this is a fine choice and it's so up to date it even covers PHP 5.0.

✓ A pocket-sized summary of PHP

✗ Not much on installation and troubleshooting

easiest way to separate these values is to use the Split method with the colon as the delimiter:

```
temp = (response.ToString()).Split
(':');
```

This returns a string array in temp, corresponding to the original response string, split into parts by each colon. From this, we extract the individual values:

```
minutes = Convert.ToInt32(temp[0]);
seconds = Convert.ToInt32(temp[1]);
frames = Convert.ToInt32(temp[2]);
```

Now we have the starting position in minutes, seconds and frames, it can be converted into a total number of frames and stored in the array and the string that we are building up:

```
offsets[i - 1] = minutes * 60 * 75 +
seconds * 75 + frames;
command.Append(offsets[i-1]+" ");
```

The last part is the trickiest, but it's not that difficult. We need the total playing time of the disc in seconds. The MCI doesn't provide this, but it can be worked out by adding the length of the last track to its starting position.

First of all, we need to get the length of the last track:

```
err = mciSendString("status cdaudio
length track " +
tracks.ToString(), response, 128,
0);
```

Next we process this to separate the minutes, seconds and frames:

```
temp = (response.ToString()).Split
(':');
minutes = Convert.ToInt32(temp[0]);
seconds = Convert.ToInt32(temp[1]);
frames = Convert.ToInt32(temp[2])
+ 1;
```

The '+1' is here because the Freedb website claims that the MCI interface returns the length of the last track minus one frame. Converting the length to frames and adding it to the starting position of the last track gives us the total playing time in frames:

```
int playtime;
playtime=offsets[tracks-1]+minutes *
60*75
+ seconds*75+frames;
```

Dividing by 75 gives the length in seconds and lets us complete the string we have been building up:

```
playtime = playtime / 75;
command.Append(playtime);
return command.ToString();
```

Try the method out using something like:

```
string c = CDAudio.getDiscData();
Console.WriteLine(c);
```

This information won't be much use to you in its raw form, but submitting it to Freedb is the key to finding the disc record stored in the database.

FREE AND EASY

The Freedb protocol is an old-fashioned 'send a command and get a reply' type. Its documentation, presumably written by someone who knows how it

all works, misses out lots of things that are obvious to them but not to us beginners. Freedb can be accessed via HTTP, SMTP or via a TCP/IP connection to port 8880. Because .NET is so good at making general TCP/IP connections, the simplest thing to do is create a TCP/IP client. Even if you're not interested in Freedb, the procedure for creating such a client is worth knowing.

Add a new class called CCDDb to the project that already has CCDAudio. We need to use the classes in System.Net.Sockets so add the following to the existing using statements:

```
using System.Net.Sockets;
```

To keep things simple, the new class will have only one method, connect, and this simply prints the progress and results to the console. In a real application, you would probably split up this single method into sub-methods and certainly do something more useful with the final data.

We need some global variables:

```
class CCDDb
{
private NetworkStream freedbstream;
private TcpClient freedb;
```

Making the TCP/IP connection is simple. All that's required is a new TcpClient object:

```
public void connect()
{
freedb= new TcpClient("freedb.freedb.
org", 8880);
Console.WriteLine( freedb.Connected);
```

The URL and port of the connection are included in the call to the TcpClient constructor. When the



constructor completes, we have a TCP/IP connection established. As before, the error handling has been ignored for reasons of space and simplicity.

GO WITH THE FLOW

To receive and send data over the TCP/IP connection, we need to retrieve the associated stream object. Stream objects are a general way of working with streams of data that can be read or written, such as disk files:

```
freedbstream = freedb.GetStream();
```

At this point we could continue and read the data sent back by the Freedb server, but this is a standard operation so it makes sense to create a private routine to do the job:

```
private string receive()
{
    byte[] buff = new byte[2000];
    freedbstream.Read(buff, 0,
        buff.GetLength(0));
    string temp=Encoding.ASCII.Get
        String(buff);
    char[] trail={'\n','\0','\r'};
    return temp.TrimEnd(trail);
}
```

This sets up a byte array to act as a data buffer, reads the stream and uses the Encoding object to convert the stream of bytes into an ASCII string.

The final part of the processing uses the TrimEnd command to remove control codes and clean up the string. With this routine defined, we can add the following line to the connect method:

```
Console.WriteLine(receive());
```

The server sends a welcome banner when you first connect, so you should see something like:

```
201 kivi CDDBP server v1.5PL2 ready at
Fri Jul 29 19:16:37 2005
```

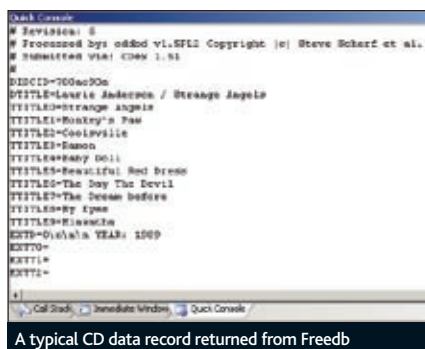
The next step is to sign on to the server using the command:

```
cddb hello username hostname
clientname version
```

Here, username is a name that identifies the user of the program, hostname is the client computer, clientname is the name of the program doing the search and version is its version number. All this information is used to find out who is using Freedb. You should try to provide realistic values for each item. To send this command, it is easier to create a routine that can be reused:

```
private void send(string message)
{
    byte[] buff = Encoding.ASCII.GetBytes
        (message+"\n");
    Console.WriteLine(message + "\n");
    freedbstream.Write(buff, 0,
        buff.GetLength(0));
}
```

The conversion from a string to an array of ASCII bytes is achieved using the Encoding object



for a second time. Without the carriage return \n added to the end of the string, the server would ignore the command. To keep track of what is happening, the same command is printed to the console and then written to the stream.

With the help of the send and receive routines, the hello command can then be sent and the reply examined:

```
send("cddb hello mike www.computer
shopper.co.uk ShopperCD 1");
Console.WriteLine(receive());
```

You should change the username and hostname to something suitable. The program identifier, shown here as ShopperCD and the version number as 1, is entirely up to you to invent for your own program. If all goes well, you should see a reply along the lines of:

```
200 Hello and welcome mike@www.computer
shopper.co.uk running ShopperCD 1
```

The 200 at the start of the line is a status/error code, and this is explained in the documentation.

LOOKING GOOD

We are now ready to look up the details of the disc currently in the drive. To do this, we first create a CCDAudio object and get the disc details:

```
CCDAudio CDAudio = new CCDAudio();
string diskdata=CDAudio.getDiscData
();
```

To look up the disc, we have to compute a disc ID based on the disc data. This can be done locally; the documentation on the website explains how. However, the Freedb server will also do the job for you via the discid command:

```
send("discid " + diskdata);
string response=receive();
Console.WriteLine(response);
```

The response from the server is in a human readable form and is something like:

```
200 Disc ID is 780ac90a
```

This can be processed using the same Split method trick, but this time splitting the string where there's a space character so the disc ID will be the fourth item:

```
string[] temp=response.Split(' ');
String DiscId=temp[4];
Console.WriteLine(DiscId);
```

Now we have everything we need to perform the lookup. We have to query the database to discover how many possible matches there are and what category of music the disc is in:

```
send("cddb query " + DiscId + " " +
diskdata);
response = receive();
Console.WriteLine(response);
```

The query simply says to send DiscId and the same disc data as before. The response can be quite complicated if there are multiple inexact matches. In this case, you have to show the user what has been found and ask them to pick the correct disc. For simplicity's sake, let's assume there is only one exact match. In this case, the response will be something like:

```
200 newage 780ac90a Laurie Anderson /
Strange Angels
```

We'll need to use the second item in the response to retrieve additional information about the CD. This can be extracted using Split:

```
temp = response.Split(' ');
String categ = temp[1];
```

Finally we can retrieve the data record that gives the track details for the disc using the read command:

```
send("cddb read " + categ + " "
+ DiscId );
Console.WriteLine(receive());
Console.WriteLine(receive());
```

In this case, all that is needed is the music category and the computed DiscId. The returned data gives the title of the disc, the computed DiscId and disk information, the track list and lots of additional data; see the website for a full range of possibilities. The data is terminated by '.' on a separate line, which is why receive is used twice. However, most users are interested only in the title and track data.

Last of all, we just have to log off Freedb and close the connection:

```
send("quit");
Console.WriteLine(receive());
freedb.Close();
}
```

If you put a call to the connect method in a button click event handler, you will discover that the details of the CD that is currently in the drive are displayed in the console window. All that remains is to parse the data so that it can be stored in your own local database.

There are other useful Freedb commands. Now that you have the basics, you can explore these by reading the documentation on the website. [ES](http://www.computershopper.co.uk)

CONTACT

MIKE JAMES

Mike has written several books on computing and programming and has been a senior lecturer at Teesside University

mike@computershopper.co.uk